

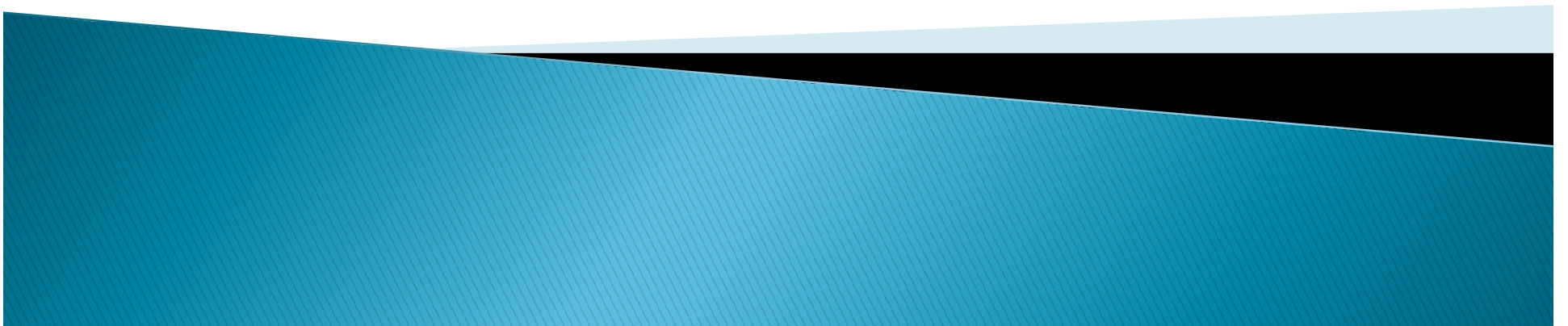
# **BCS6B13: Fundamentals of Operating Systems**

## **Module 2**

Process / Thread / Semaphore / Deadlocks

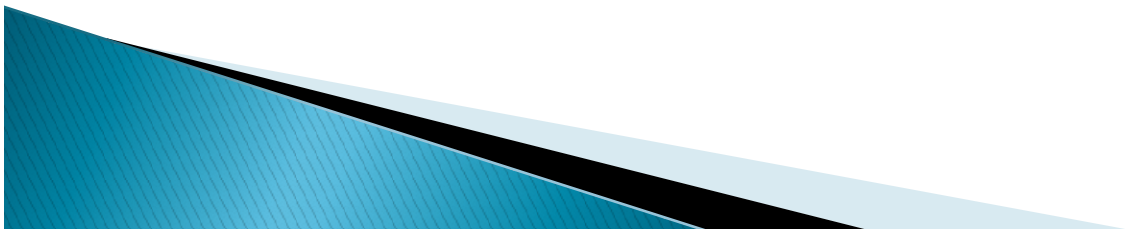
Ms.Hitha Paulson  
Assistant Professor, Dept of Computer Science  
Little Flower College, Guruvayoor

# Process



# Process Concept

- ▶ An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- ▶ uses the terms *job* and *process* almost interchangeably
- ▶ **Process – a program in execution**



# The Process

## ▶ Multiple parts

- The program code, also called text section
- Current activity including program counter, processor registers
- Stack containing temporary data
  - Function parameters, return addresses, local variables
- Data section containing global variables
- Heap containing memory dynamically allocated during run time

## ▶ Program is passive entity, process is active

- Program becomes process when executable file loaded into memory

## ▶ Execution of program started via GUI mouse clicks, command line entry of its name, etc

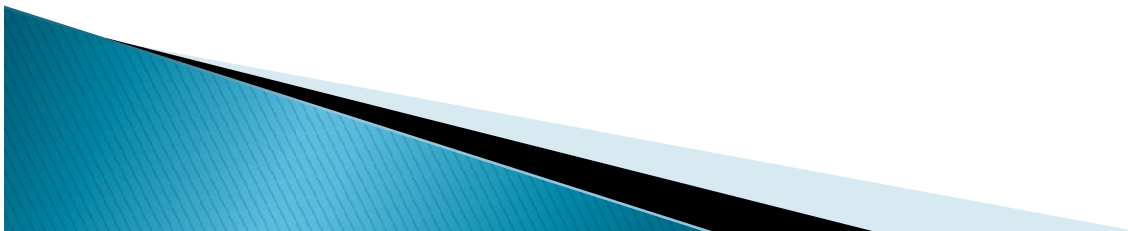
## ▶ One program can be several processes

Consider multiple users executing the same program

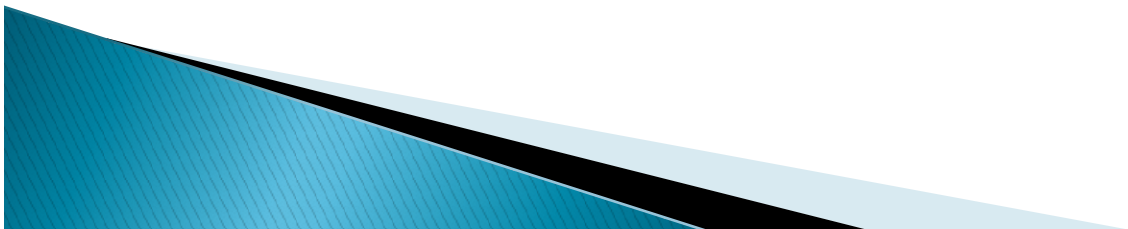
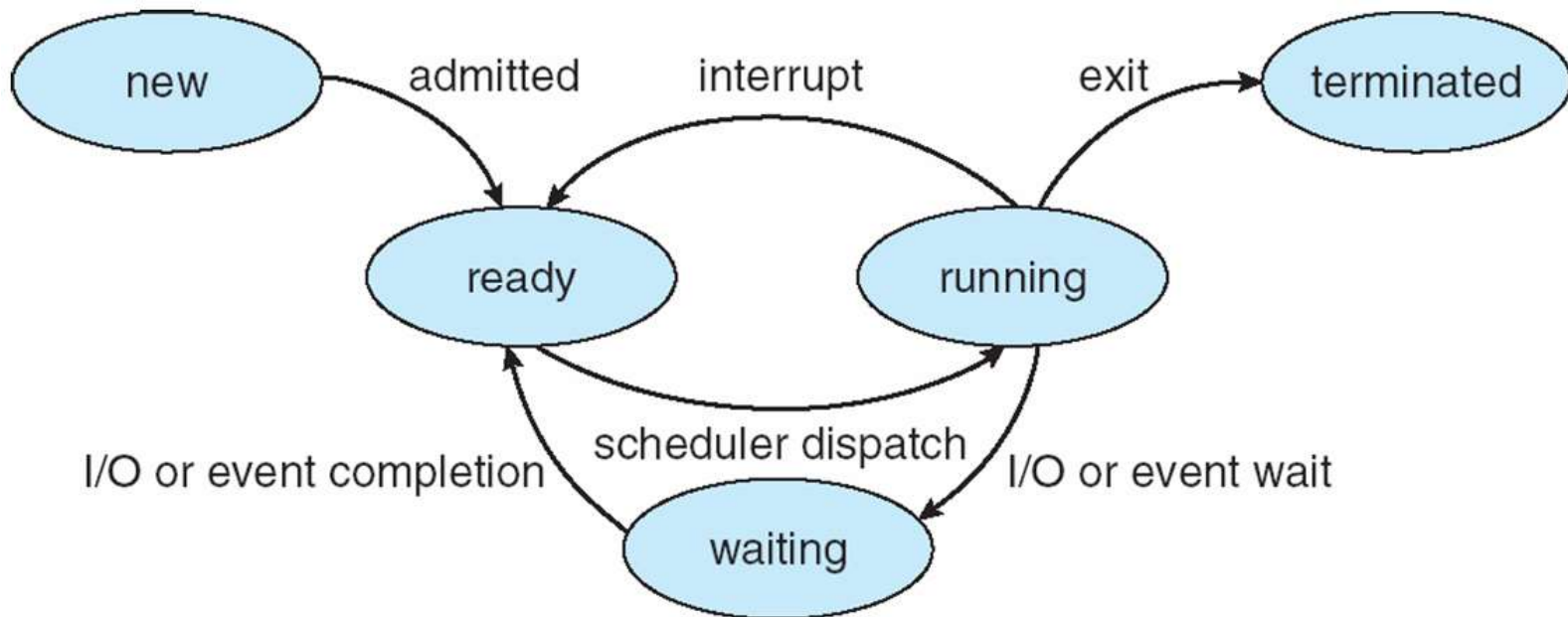


# Process State

- ▶ As a process executes, it changes *state*
  - new: The process is being created
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - ready: The process is waiting to be assigned to a processor
  - terminated: The process has finished execution



# Diagram of Process State



# Process Control Block (PCB)

- ▶ Each Process is represented in the operating system by a Process Control Block (PCB) also called Task control Block.

It contains Information associated with each process

1. Process state
2. Program counter
3. CPU registers
4. CPU scheduling information (process priority, pointer to scheduling queue etc...)
5. Memory-management information (base and limit reg value, page table etc..)
6. Accounting information (amount of CPU and real time used, time limit, process no etc...)
7. I/O status information (list of IO devices allocated, open files etc..)

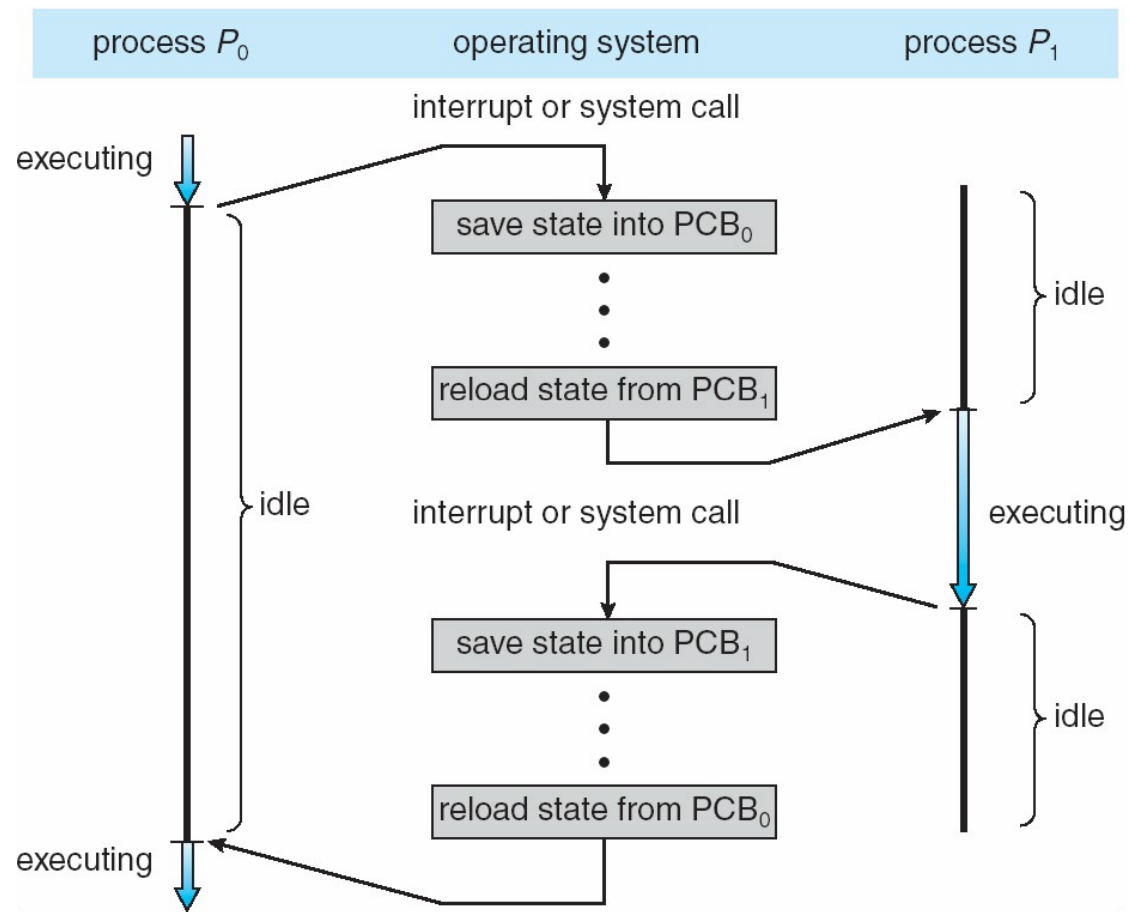


# Process Control Block (PCB)



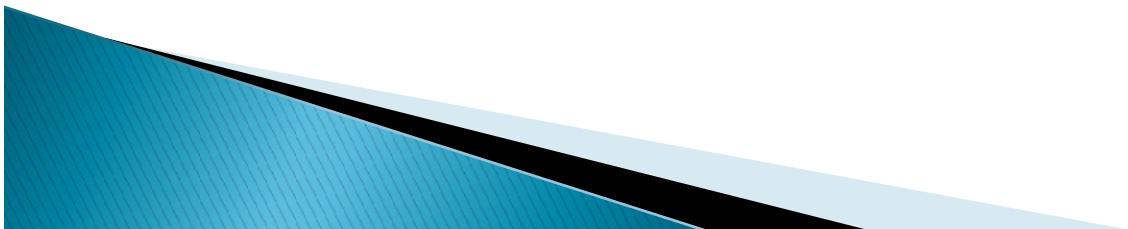


# CPU Switch From Process to Process

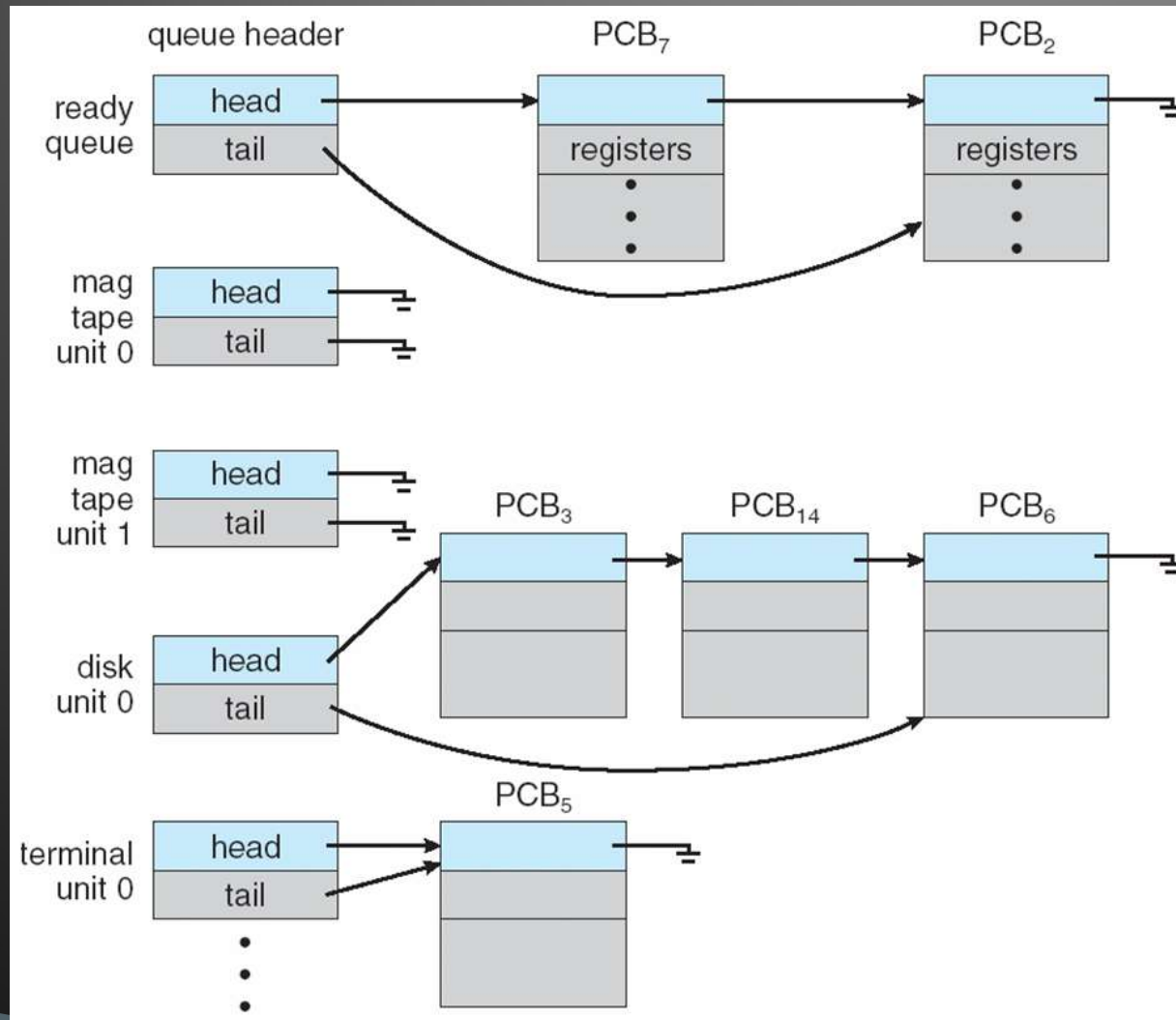


# Process Scheduling

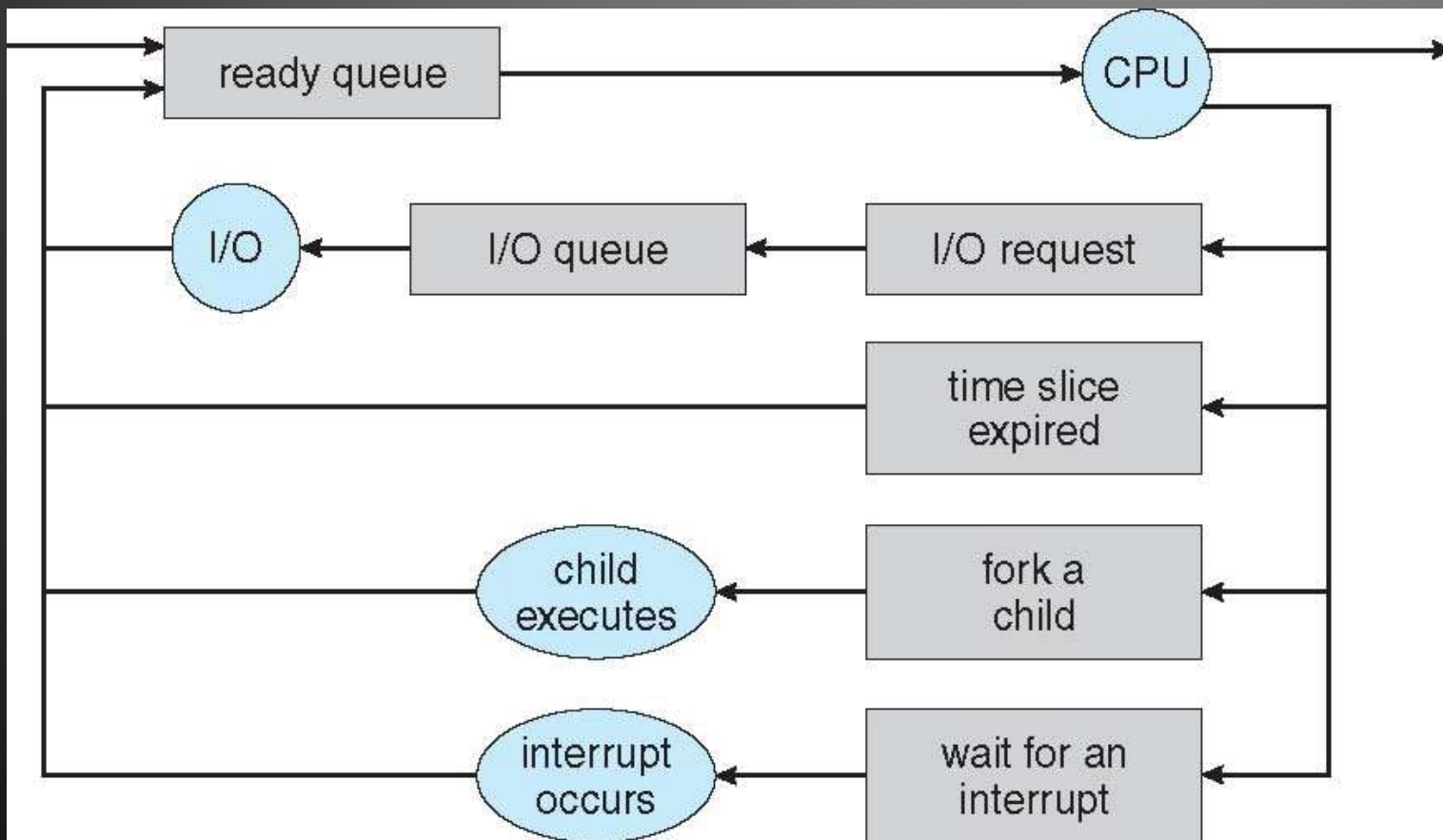
- ▶ Maximize CPU use, quickly switch processes onto CPU for time sharing
- ▶ Process scheduler selects among available processes for next execution on CPU
- ▶ Maintains scheduling queues of processes
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute (linked list). Ready queue contains pointer to the first and final PCBs in the list.
  - Device queues – set of processes waiting for an I/O device
  - Processes migrate among the various queues



# Ready Queue And Various I/O Device Queues

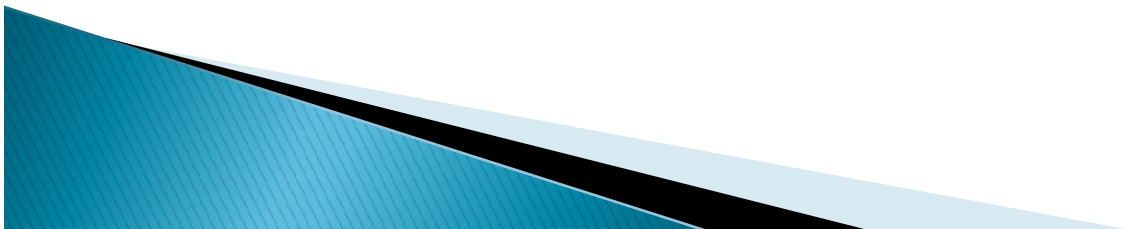


# Representation of Process Scheduling



# Schedulers

- ▶ **Long-term scheduler** (or job scheduler) – selects which processes should be brought into memory for execution.
- ▶ **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system



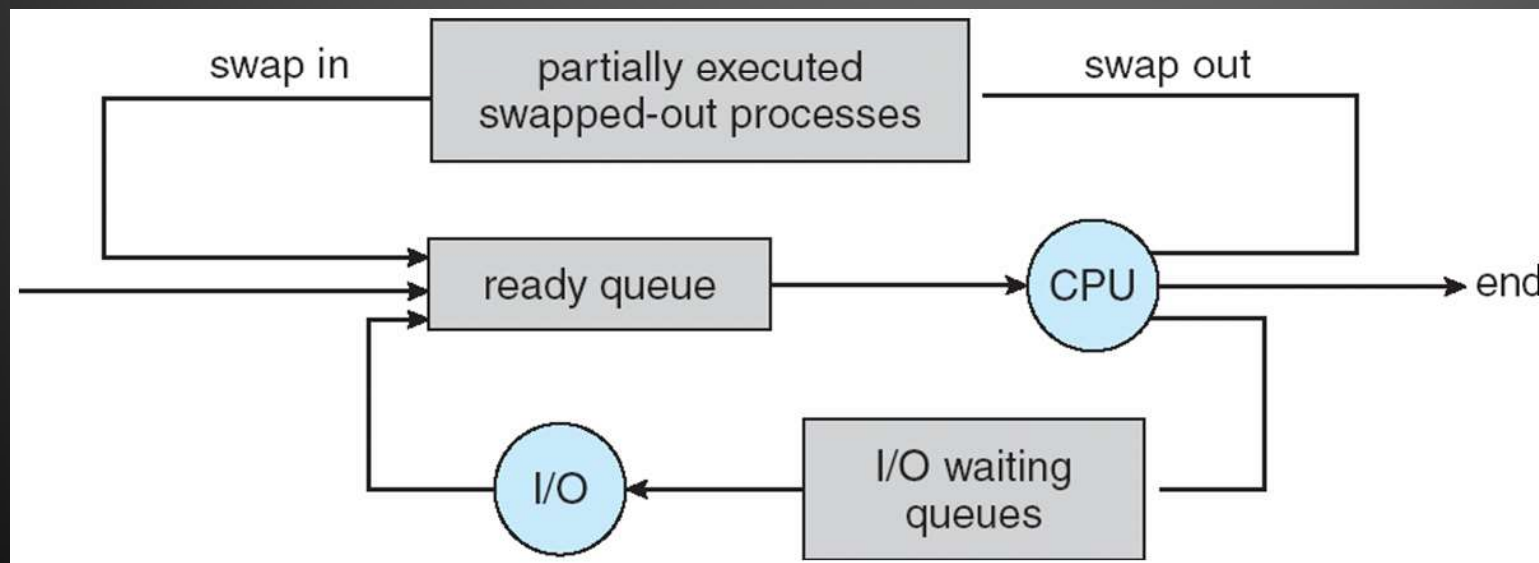
# Schedulers

- ▶ Short-term scheduler is invoked very frequently (milliseconds)  $\Rightarrow$  (must be fast)
- ▶ Long-term scheduler is invoked very infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
- ▶ The long-term scheduler controls the *degree of multiprogramming*
- ▶ *Long term scheduler is invoked when a process leaves the system*
- ▶ Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations
  - CPU-bound process – spends more time doing computations

The long term scheduler should select a good process mix of IO bound and CPU bound processes

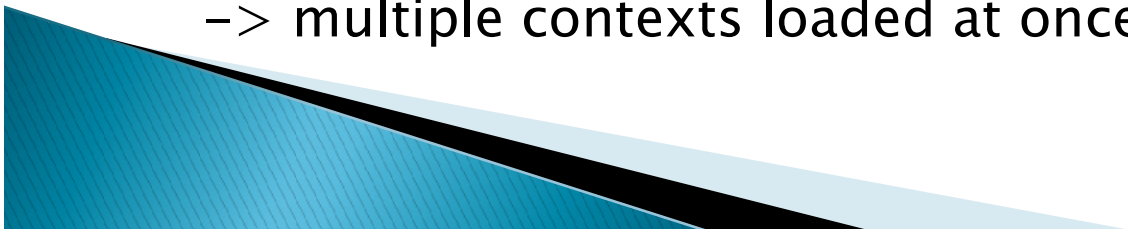
# Addition of Medium Term Scheduling

It removes processes from memory and thus reduces the degree of multi programming



# Context Switch

- ▶ When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a **context switch**.
- ▶ **Context** of a process represented in the PCB
- ▶ Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → longer the context switch
- ▶ Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once

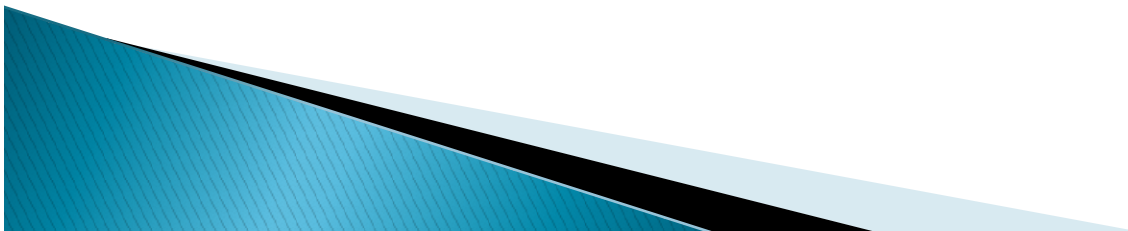




# Operations on Operating Systems

Process Creation

Process Termination



# Process Creation

- ▶ Parent process create children processes, which, in turn create other processes, forming a tree of processes
- ▶ Generally, process identified and managed via a process identifier (pid)
- ▶ Possibilities in Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share partition the resources
  - Share resources among some of the processes.
- ▶ Possibilities in Execution
  - Parent and children execute concurrently (*asynchronous Process Creation*)
  - Parent waits until children terminate (*synchronous Process Creation*)

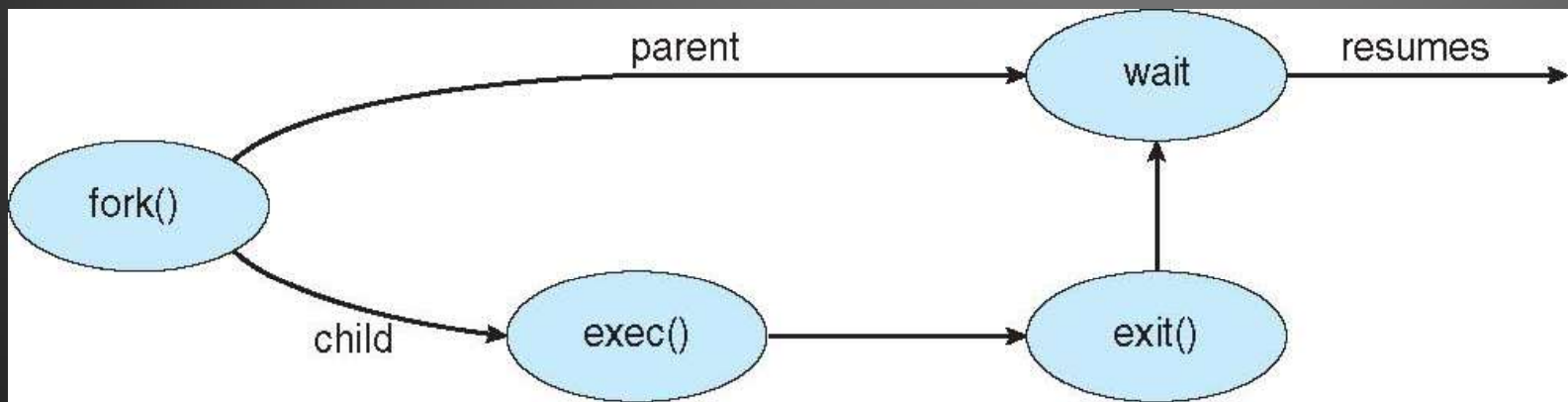
# Process Creation (Cont.)

- ▶ Possibilities in Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- ▶ Windows – CreateProcess() system call
- ▶ UNIX
  - fork system call creates new process in same address space
  - execlp system call used after a fork to replace the process' memory space with a new program

**The task of creating a new process on the request of some other process is called Process Spawning**



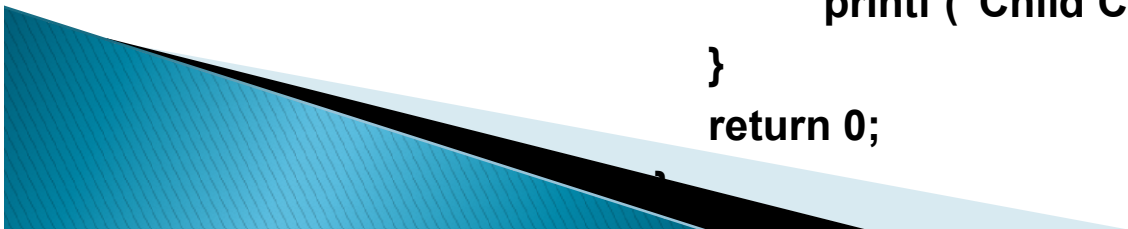
# Process Creation



# C Program Forking Separate Process in Unix

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
    int pid;

    pid = fork(); /* fork another process */
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execlp ("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child */
        wait (NULL);
        printf ("Child Complete");
    }
    return 0;
}
```



# Process Termination

- ▶ Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- ▶ Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated if parent terminates– **cascading termination**



# Cooperating Processes

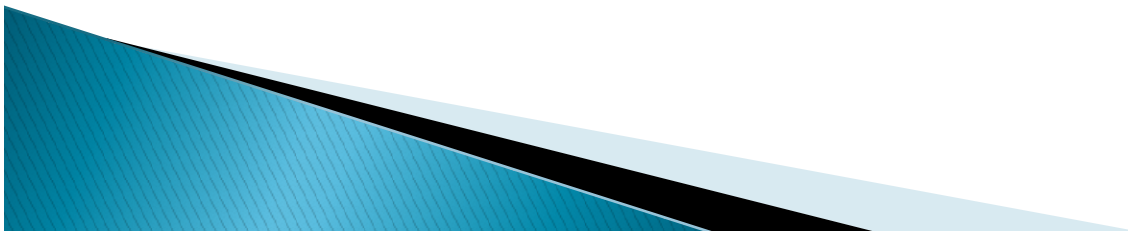
- ▶ Processes within a system may be independent or cooperating
- ▶ Cooperating process can affect or be affected by other processes, including sharing data
- ▶ Reasons for cooperating processes:
  - **Information sharing** (many interested in same data)
  - **Computation speedup**(a particular task to run faster divide to sub tasks and execute in parallel in multiple parallel components)
  - **Modularity** (to construct system in modular fashion)

**Convenience**



# Cooperating Processes

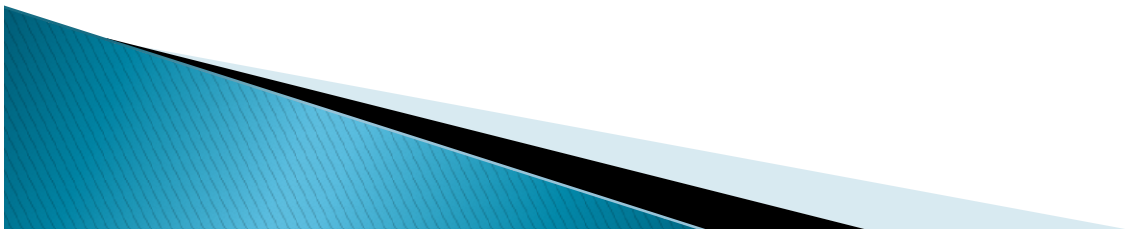
- ▶ Independent process cannot affect or be affected by the execution of another process
- ▶ Cooperating process can affect or be affected by the execution of another process





# Communication Techniques

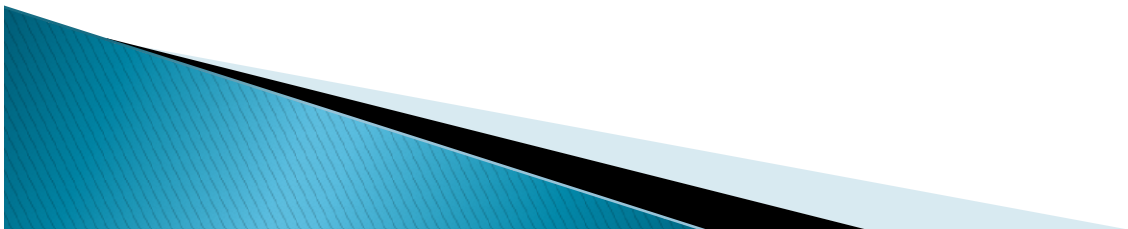
- ▶ Cooperating Processes communicate through
- ▶ **Inter process communication (IPC)**
  - **Shared Memory** (*Share same address space*)
  - **Message Passing** (*when processes is in different computers in a network*)



# Producer–Consumer Problem

▶ Shared Memory

- ▶ Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size
- ▶ Buffer either provided by Operating system through IPC or by the application Programmer.



# Bounded-Buffer – Shared-Memory Solution

## ▶ Shared data

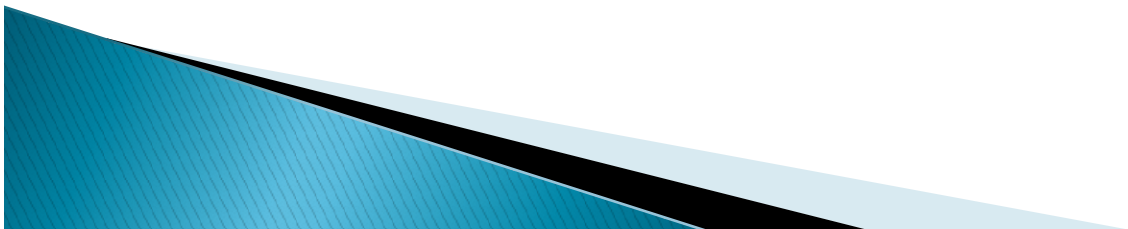
```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- ▶ Circular array
- ▶ In points to next free position.
- ▶ Out points to first full position.
- ▶ Next location =  $(in+1)\%buffersize$
- ▶ Buffer empty :  $in == out$
- ▶ Buffer full :  $(in+1) \% buffersize == out$

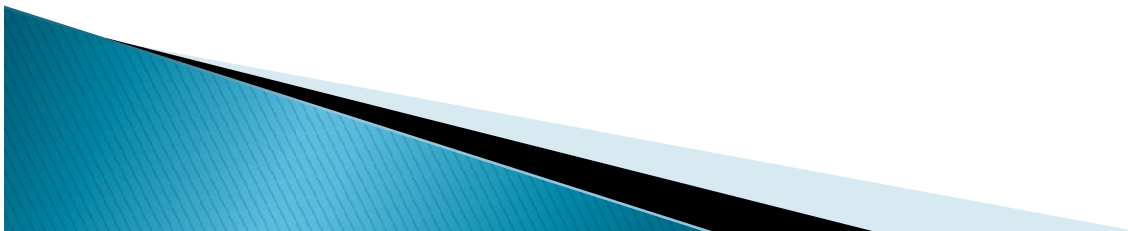
# Bounded-Buffer – Producer

```
while (true) {  
    /* Produce an item */  
    while (((in = (in + 1) % BUFFER SIZE count)  
== out)  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```



# Bounded Buffer – Consumer

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to  
    consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```

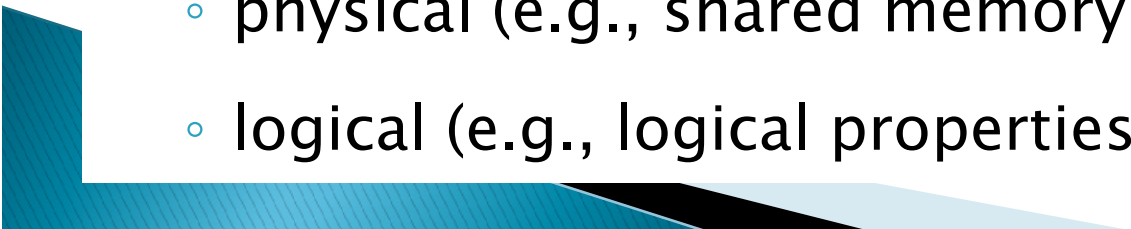


## Inter process Communication –Message Passing

- ▶ Mechanism for processes to communicate and to synchronize their actions
- ▶ Message system – processes communicate with each other without resorting to shared variables

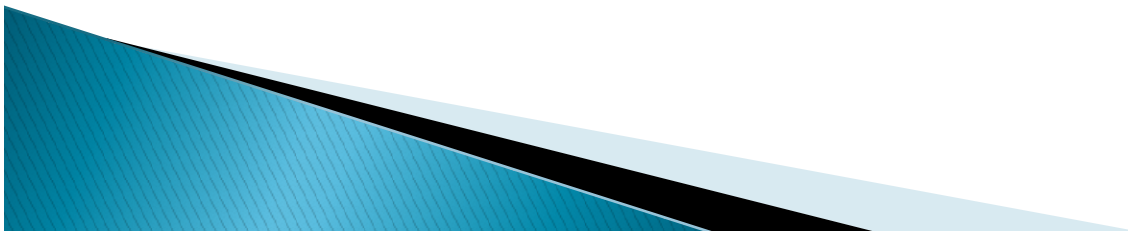


# Interprocess Communication – Message Passing

- ▶ IPC facility provides two operations:
    - `send(message)` – message size fixed or variable
    - `receive(message)`
  - ▶ If  $P$  and  $Q$  wish to communicate, they need to:
    - establish a *communication link* between them
    - exchange messages via send/receive
  - ▶ Implementation of communication link
    - physical (e.g., shared memory, hardware bus)
    - logical (e.g., logical properties)
- 

# Logical implementation of send/receive operation

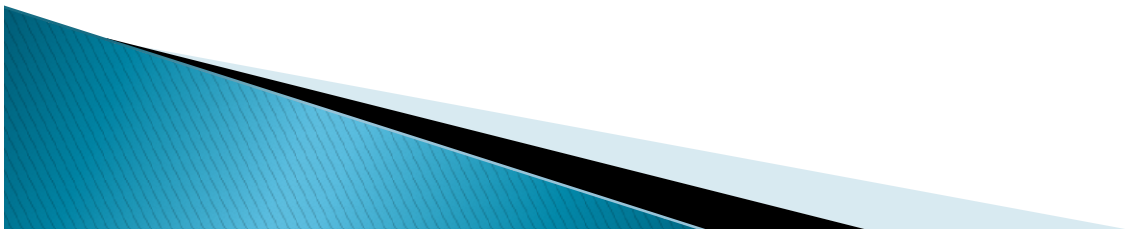
1. Direct or indirect communication
2. Symmetric or asymmetric communication
3. Automatic or explicit buffering
4. Send by copy or send by reference
5. Fixed size or variable sized message.





# Direct –Symmetric Communication

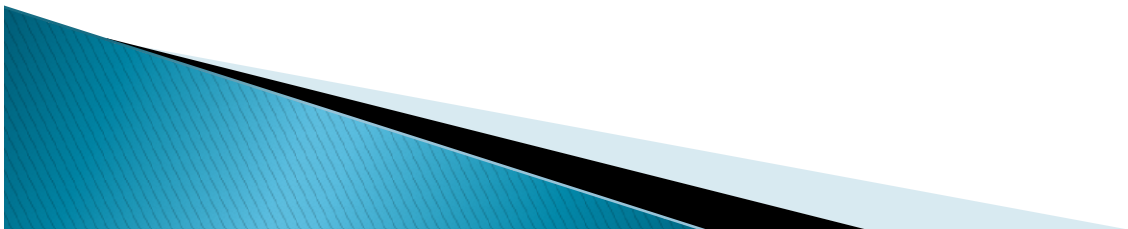
- Processes must name each other explicitly:
  - `send (P, message)` – send a message to process P
  - `receive(Q, message)` – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional



# Direct –Asymmetric Communication

- Only sender names the recipient
  - **send** ( $P$ , *message*) – send a message to process  $P$
  - **receive**(*id*, *message*) – receive message from any process , id set to the name of the process.

Disadvantage, Direct :Change in name results in examining all process definitions



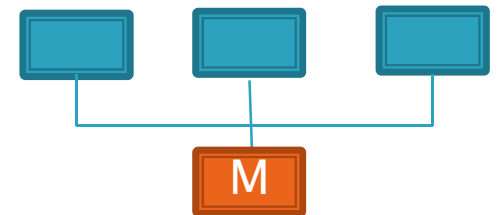
# Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

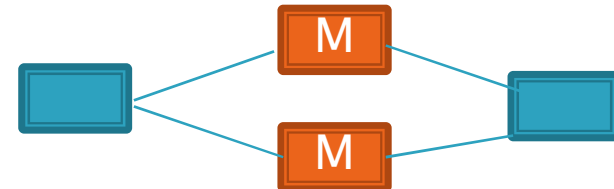
- Properties of communication link

- Link established only if processes share a common mailbox

- A link may be associated with many processes



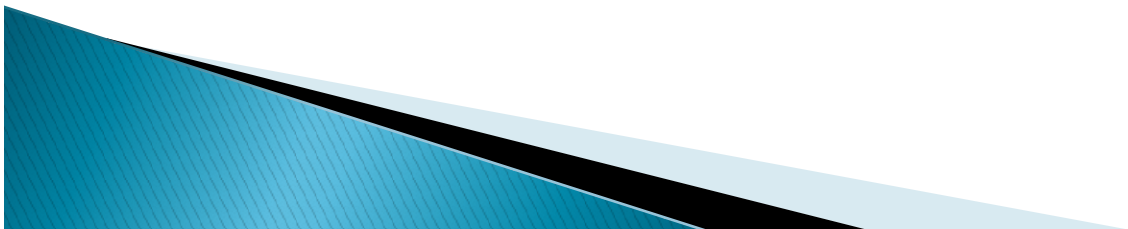
- Each pair of processes may share several communication links



- Link may be unidirectional or bi-directional

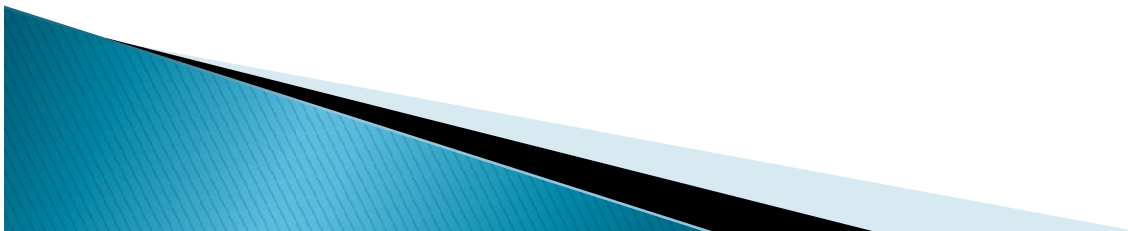
# Indirect Communication

- Mailbox sharing
  - $P_1$ ,  $P_2$ , and  $P_3$  share mailbox A
  - $P_1$ , sends;  $P_2$  and  $P_3$  receive
  - Who gets the message?
- Depends on the system scheme...
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.



# Mailbox Owner

- ▶ Owned by Operating System or by process
- ▶ If owned by Process mailbox terminates along with process.
- ▶ If owned by OS mailbox not attached to any process.



# Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** has the sender send the message and continue
  - **Non-blocking receive** has the receiver receive a valid message or null

■ Different combinations possible

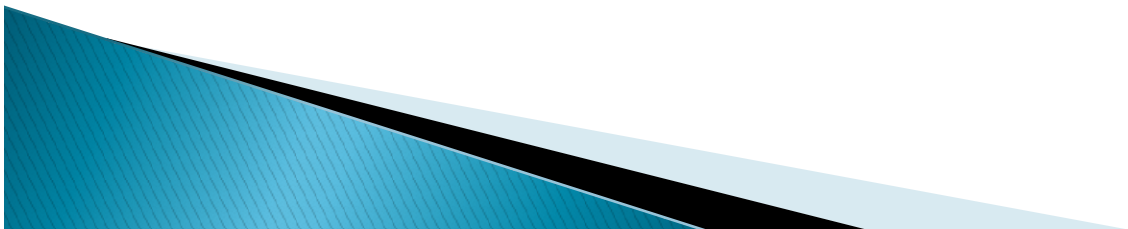
- If both send and receive are blocking, we have a **rendezvous**

**Rendezvous: meet at an agreed time and place**

# Buffering

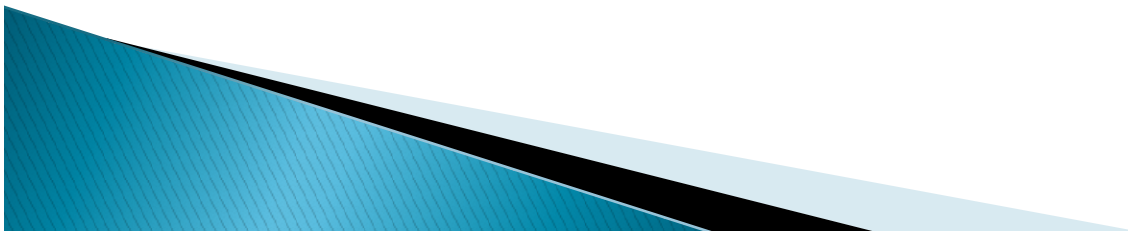
- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages  
Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of  $n$  messages  
Sender must wait if link full
  3. Unbounded capacity – infinite length  
Sender never waits

Case 2 and 3 known to be automatic buffering



# Communication in Client Server systems

- ▶ Sockets
- ▶ Remote Procedure Calls
- ▶ Remote Method Invocation





# Communication in Client Server systems

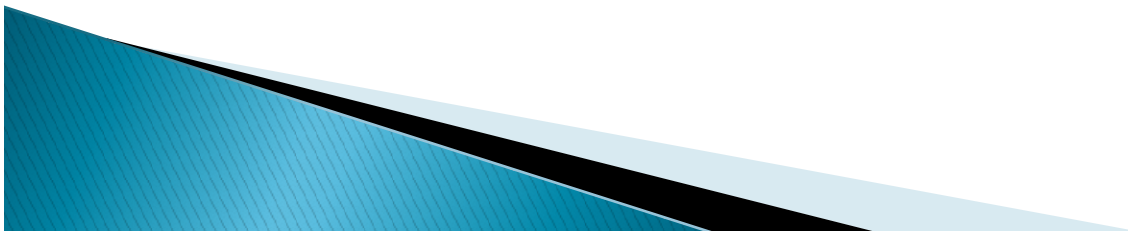
## ▶ Sockets

- ▶ A **socket** is defined as an endpoint for communication
- ▶ Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- ▶ The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- ▶ Communication consists between a pair of sockets
- ▶ All ports below 1024 are *well known*, used for standard services
- ▶ When client initiates a request , it is assigned a port number by the host computer which is greater than 1024



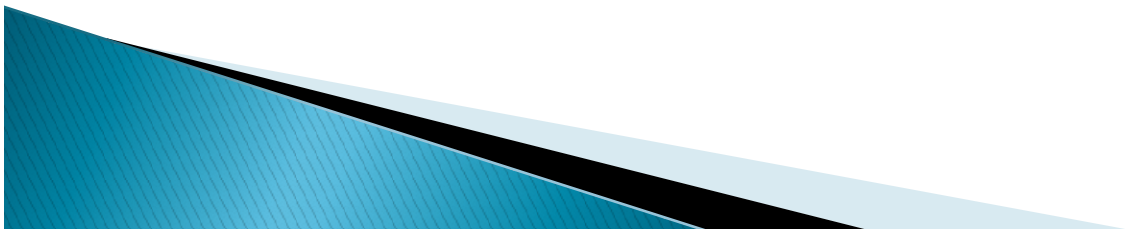
# ▶ Three types of sockets

- ▶ Connection-oriented (TCP)
- ▶ Connectionless (UDP)
- ▶ MulticastSocket class- data can be sent to multiple recipients



# Communication in Client Server systems

- ▶ Communication using sockets common and efficient, but low level form of communication.
- ▶ Unstructured data.
- ▶ It is the responsibility of the client server application to impose structure on data.
- ▶ Alternative.....Higher level methods of communication
  - **RPC**
  - **RMI**



# Remote Procedure calls

- ▶ Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
- ▶ Messages are well structured.
- ▶ Stubs – client–side proxy for the actual procedure on the server
- ▶ The client–side stub locates the server and marshalls the parameters
- ▶ The server–side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server.



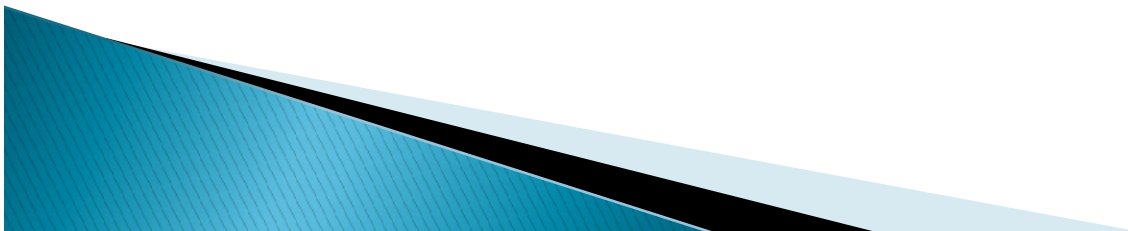
# Remote Procedure calls

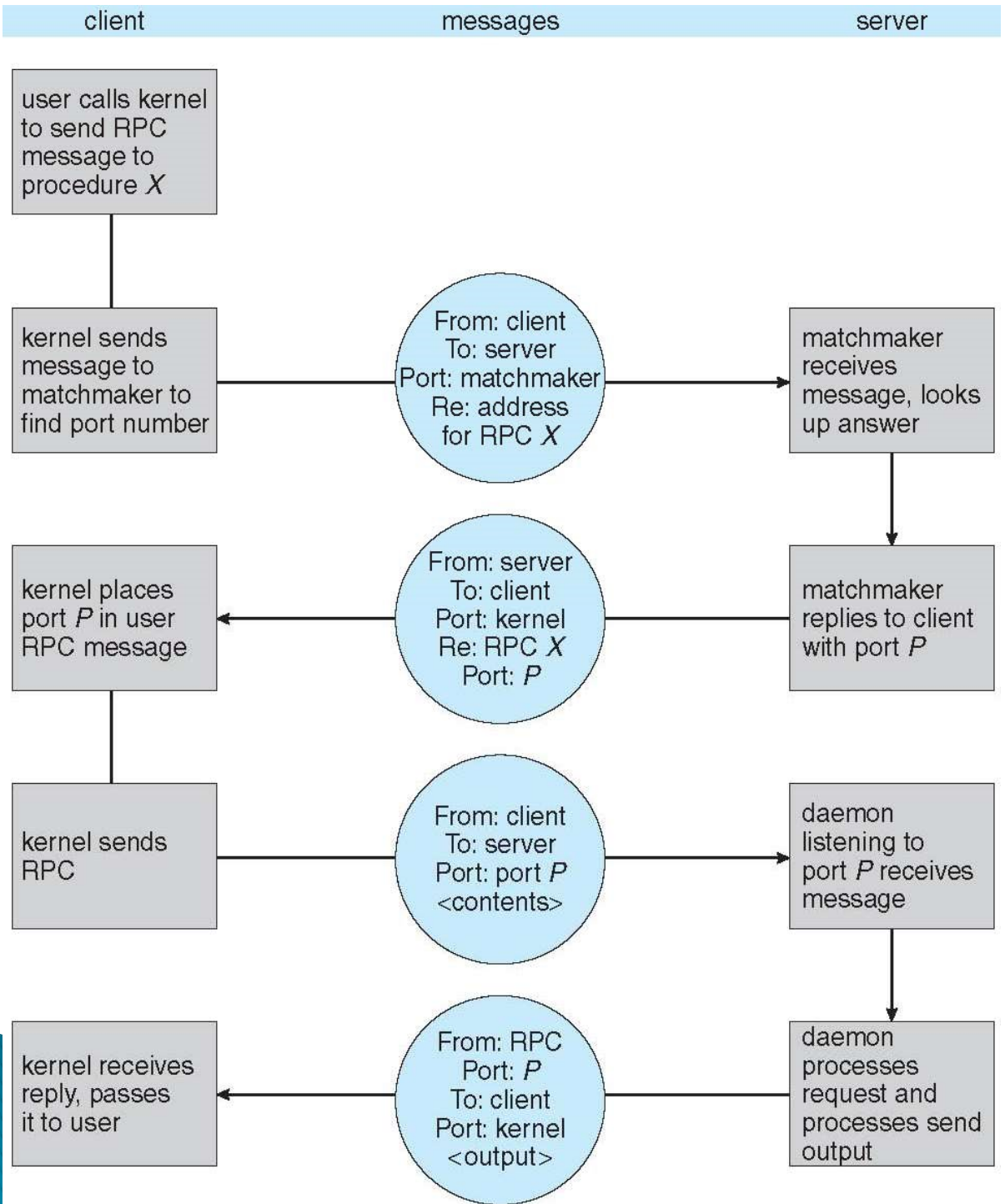
- ▶ Data representation handled via **External Data Representation (XDR)** format to account for different architectures
- ▶ Marshalling involves converting machine dependent data to XDR
- ▶ Remote communication has more failure scenarios than local
  - Messages can be delivered *exactly once* rather than *at most once*



# Remote Procedure calls – Binding

- ▶ Binding: procedure calls name is replaced by the memory address of the procedure call.
- ▶ Binding in RPC?
  - Predetermined binding, fixed port address.
  - Matchmaker technique

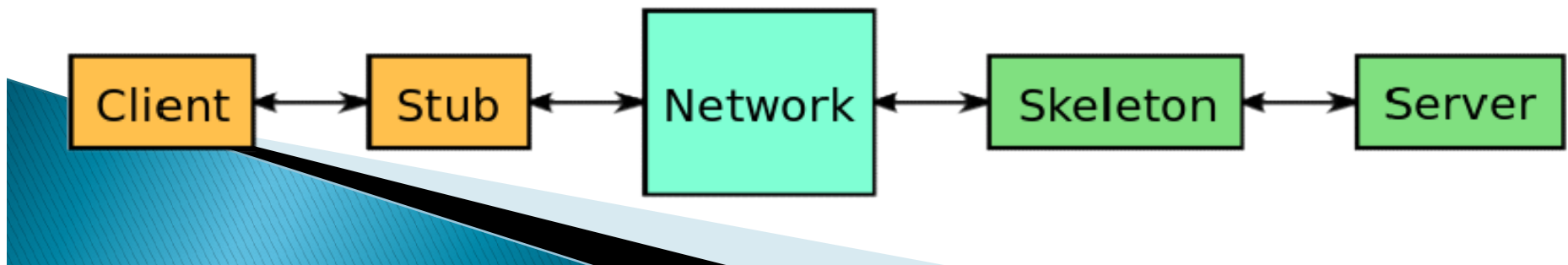




# Matchmaker Technique

# Remote Method Invocation

- ▶ Java feature similar to RPC
- ▶ Invoke method on a remote object.
- ▶ Remote Object: Objects in different JVM
- ▶ RPC support procedural programming
- ▶ RMI support object oriented programming
  - RMI can pass objects as parameters





# Remote Method Invocation

- ▶ If the marshaled parameters are local or non remote they are passed by copy using a technique called object serialization.
- ▶ If the parameters are remote they are passed by reference.

